ECE 438: Computer Design
Electrical and Computer Engineering
University of New Mexico

Instructor: Andrew Targhetta

Lab 4

A Pipelined MIPS Processor, Part I

Due: April 27th, 2020

Introduction

The objective of lab this week is to develop a couple of key components necessary for our pipelined MIPS processor, namely the pipeline control unit and the forwarding unit. Previously we developed a single-cycle MIPS processor with a register file, ALU, ALU decoder, and Instruction decoder, and we will reuse those components in our pipelined design.

Figure 1 shows a block diagram of the 5-stage, pipelined MIPS processor you will design in lab. As shown, your processor will use forwarding to resolve data hazards and separate memories to resolve structural hazards. Control hazards will be resolved using branch delay slots in combination with the assume Not Taken branching scheme discussed in lecture.

Background

In lecture, it was shown that Read After Write (RAW) data hazards involving ALU instructions, such as ADD, SUB, AND, etc., can be completely resolved by forwarding in the classic five-stage processor pipeline. The premise is that the result for an ALU instruction, in the case of a RAW with another ALU instruction, already exists in the pipeline but has not yet been committed to the register file. In Figure 1, you will notice that the ALU source multiplexers have four inputs, two of which facilitate forwarding by pulling data from the Memory and Write-back stages ahead of the Execute stage. The job of the forwarding

unit found in the decode stage is to detect data hazards that are resolvable via forwarding and produce the control signals for the ALU source multiplexers appropriately.

When a load instruction produces a value that is consumed by the instruction immediately following in the pipeline, a stall is unavoidable. This is due to the fact that the value from the load is produced in the Memory stage rather than the Execute stage. Thus, the pipeline control unit is required to identify and resolve data hazards that cannot be completely fixed with bypass logic. In the case of a load as a producer in a RAW hazard, the pipeline control unit stalls the fetch and decode stages for one clock cycle. The aforementioned technique for resolving data hazards is commonly referred to as pipeline interlocking. Two additional multiplexers are found in the diagram of our MIPS processor for forwarding register values to the data memory. In the case of a RAW between a store following a load, the bypass logic is able to completely avoid a stall. Naturally, this is a result of the fact that the store does not need the register value until the beginning of the Memory stage.

In addition to resolving data hazards, the pipeline control unit is responsible for orchestrating pipeline flushes due to control hazards (i.e. jumps and branches). Control hazards due to branching come from the fact that the processor fetches in the first stage, while it does not resolve the branch until the third stage (second or fourth in other designs). If the branch logic is moved to the second (Decode) stage, the penalty for a branch is only one clock cycle. However, bypass logic must also be created for the conditional evaluation, which is why Figure 1 shows that the branch logic in the third (Execute) stage.

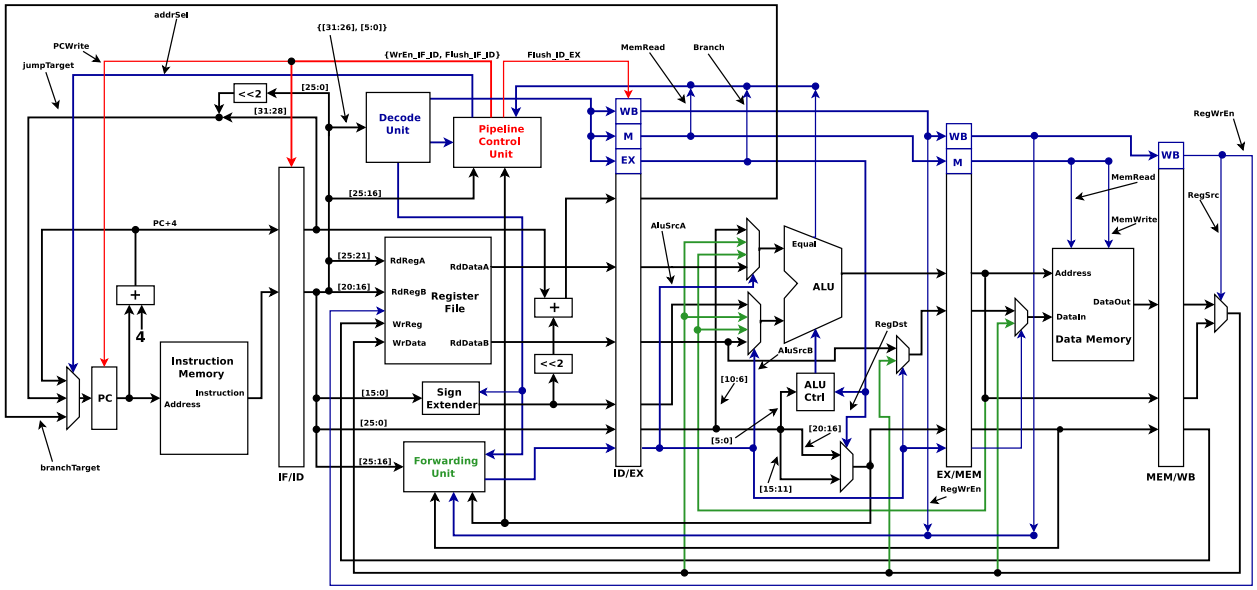

Figure 1: MIPS Block Diagram

Procedure

1. Describe the **Forwarding Unit** in VHDL. The forwarding unit is responsible for generating the correct selection bits for the ALU source multiplexers, as well as the data memory forwarding multiplexers. In our pipeline, the forwarding unit is in the decode stage, which means it must compare the *WrReg* and *RegWrEn* signals in the execute and memory stages to the *Rs* and *Rt* fields of the instruction being decoded. Use the following VHDL code as a starting point:

```
1  ----------------------------------------------------
   -- This component describes the forwarding unit in our
3  -- 5-stage, pipelined MIPS processor.
   ----------------------------------------------------
5
   library ieee;
7  use ieee.std_logic_1164.all;

9  ----------------------------------------------------

11 entity ForwardingUnit is

13 port( UseShamt, UseImmed: in std_logic;
         EX_RegWrEn, MEM_RegWrEn: in std_logic;
15       ID_Rs, ID_Rt: in std_logic_vector(4 downto 0);
         EX_WrReg, MEM_WrReg: in std_logic_vector(4 downto 0);
17       AluSrcA, AluSrcB: out std_logic_vector(1 downto 0);
         DataMemForwardCtrl_EX: out std_logic;
19       DataMemForwardCtrl_MEM: out std_logic
   );
21 end ForwardingUnit;

23 ----------------------------------------------------

25 architecture behv of ForwardingUnit is

27 begin
   ----------------------------------------------------
29 --forwarding logic for ALU operand A

31 --insert code here!

33 ----------------------------------------------------
   --forwarding logic for ALU operand B
35     --The sensitivity list! Do something if any of these signal change
       process(UseImmed, ID_Rt, EX_WrReg, MEM_WrReg, EX_RegWrEn, MEM_RegWrEn)
37     begin --priority is important!
       --If immediate instruction, don't forward
39     if (UseImmed='1') then
           AluSrcB <= "00";--use immediate
```

```
41      --else if data is produced by instruction one stage ahead
        elsif ((EX_RegWrEn='1') and (ID_Rt=EX_WrReg) and (EX_WrReg/=b"00000")) then
43          AluSrcB <= "10";--use EX bypass
        --else if data is produced by instruction two stages ahead
45      elsif ((MEM_RegWrEn='1') and (ID_Rt=MEM_WrReg) and (MEM_WrReg/=b"00000"))
            then
            AluSrcB <= "01";--use MEM bypass
47      else
        --else if data is produced by instruction in WB or beyond
49          AluSrcB <= "11";--pull from Register file
        end if;
51      end process;
    ---------------------------------------------------
53  --forwarding logic for data memory

55  --insert code here!

57

    end behv;
59
    ---------------------------------------------------
```

2. Describe the **Pipeline Control Unit** in VHDL. The pipeline control unit orchestrates the flow of instructions through the pipeline by controlling the stalling and flushing of the pipeline registers as well as selecting the correct next *PC* address. The entity for the pipeline control unit is provided below:

```
    ---------------------------------------------------
2   -- This component describes the pipeline control unit
    -- in our 5-stage, pipelined MIPS processor.
4   ---------------------------------------------------

6   library ieee;
    use ieee.std_logic_1164.all;
8
    entity PipelineCtrl is
10
    port( EX_Branch: in std_logic;
12        EX_Equals: in std_logic;
          ID_Jump: in std_logic;
14        ID_Rs, ID_Rt: std_logic_vector(4 downto 0);
          UseShamt, UseImmed: in std_logic;
16        EX_MemRdEn: in std_logic;--to detect a load
          EX_WrReg: in std_logic_vector(4 downto 0);
18        PCwrite: out std_logic;
          addrSel: out std_logic_vector(1 downto 0);
20        Flush_IF_ID, WrEn_IF_ID: out std_logic;
          Flush_ID_EX: out std_logic
```

```
22 );
   end PipelineCtrl;
```

Specifically, when a "Taken" branch instruction is detected in the execute stage, the pipeline control unit must flush the instruction in the fetch stage and resume fetching at the branch target address. Remember that we do *not* want to flush the instruction immediately following the branch in program order. This means that we allow the instruction in decode to progress through the pipeline. Similarly, when a jump instruction is detected in the decode stage, the pipeline control unit must simply redirect fetching to the jump address. Note that the jump instruction also includes a delay slot. In other words, the instruction immediately following the jump is executed. Finally, the pipeline control unit must detect and interlock the processor when a data hazard that cannot be resolved with forwarding exists. In the case of our simple pipeline, the only data hazard that cannot be resolved with forwarding alone is when an instruction needs a value from a load immediately ahead of it in the pipeline. When such a hazard exists, the pipeline control unit must simply stall the fetch and decode stages, allowing the load to proceed down the pipeline. Stalling only the front end of the pipeline is often times referred to as a pipeline *slip*.

1 Deliverables

1. Email all VHDL source files with comments to `adtargh@unm.edu` or submit via the Learn system .

   Note: Code submitted without adequate comments will not be graded!

2. Answer the following questions:

   (a) In our pipeline, can the data hazard below be resolved with forwarding alone? Why or why not?

   ```
   lw $t0, 0($t1)
   sw $t0, 0($t2)
   ```

   (b) Will the pipeline control unit *slip* the pipeline in the case of the instruction sequence above? Why or why not?